LBL-30056
UC-404

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

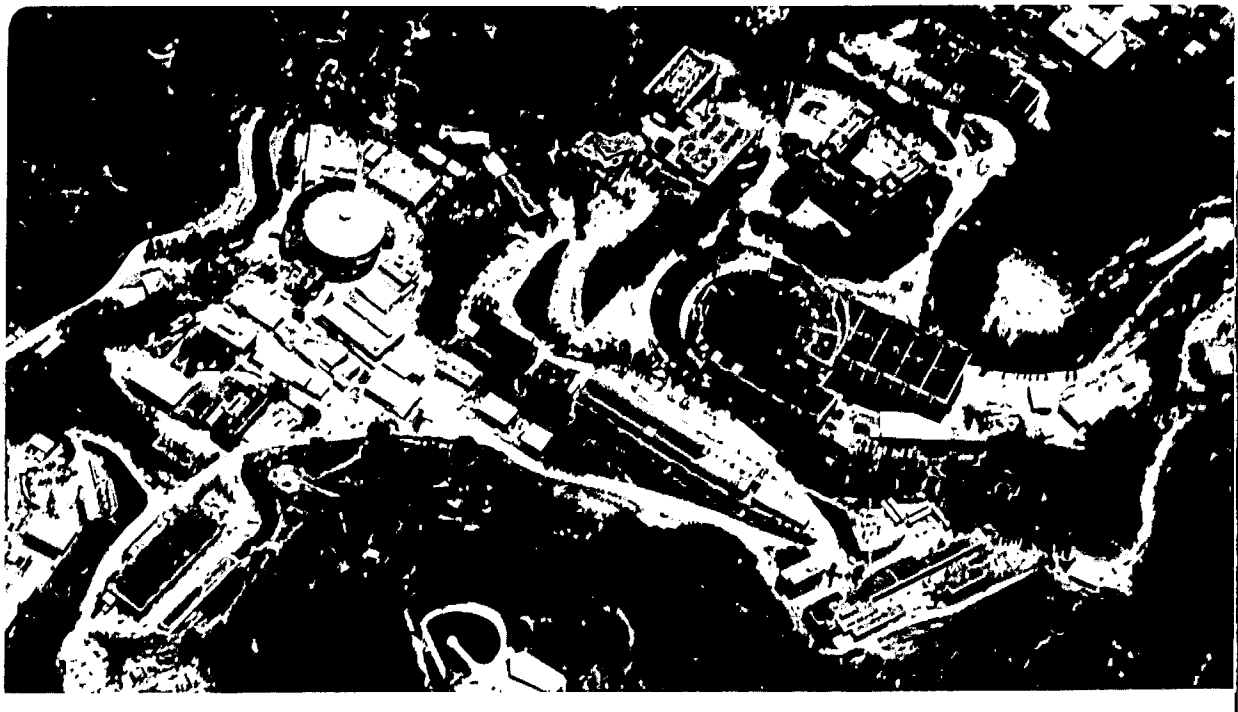
Physics Division

Mathematics Department

Canonical vs. Micro-Canonical Sampling Methods in a 2D Ising Model

J. Kepner

December 1990



LOAN COPY
Circulates
for 2 weeks
Bldg. 50 Library.
LBL-30056
Copy 2

Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

LBL-30056

CANONICAL VS. MICRO-CANONICAL SAMPLING METHODS IN A 2D ISING MODEL[1]

Jeremy Kepner
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

December 1990

[1] This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

KEYWORDS

2D Ising Model, Micro-Canonical, Specific Heat

ABSTRACT

Canonical and micro-canonical Monte Carlo algorithms were implemented on a 2D Ising model. Expressions for the internal energy, U, inverse temperature, Z, and specific heat, C, are given. These quantities were calculated over a range of temperatures, lattice sizes, and time steps. Both algorithms accurately simulate the Ising model. To obtain greater than three decimal accuracy from the micro-canonical method requires that the more complicated expression for Z be used. The overall difference between the algorithms is small. The physics of the problem under study should be the deciding factor in determining which algorithm to use.

1. INTRODUCTION

In the study of phase transitions and other critical phenomena the Monte Carlo method has emerged as one of the most powerful simulation techniques—canonical and micro-canonical being just two of the more common approaches. Determining which is best for a particular problem can be difficult. It is the goal of this paper to help the researcher by comparing the ability of these two algorithms to simulate a 2D Ising model, with an emphasis on illustrating the differences in behavior and accuracy as a function of temperature and lattice size.

In the canonical approach, one computes the state of each point from the previous state using a random number generator. Usually, each state is weighted according to a probability proportional to the Boltzman factor exp(–E/kT), with E being the energy of the state [1].

The micro-canonical Monte Carlo method consists of constraining the total energy of the system, while letting the energy distribution evolve. The transfer of energy is carried out by a new set of variables, called demons, which correspond to the kinetic energy in molecular dynamics calculations [2].

2. THE UPDATING ALGORITHMS

In both algorithms, the central operations are performed on a lattice of spins, S, that are either up (+1) or down (-1). For our model, we will define the energy of a particular site i as

$$E_i = \sum(1 - S_i S_j), \tag{1}$$

where the sum is over the four nearest neighbors j of i [3]. This has the operational effect of making a lattice with aligned spins have lower energy

than an unaligned lattice. Note that flipping the spin of i produces a change in energy

$$\Delta E_i = 2\sum S_i S_j. \tag{2}$$

A quantity of particular interest in our experiments is the internal energy, U. In terms of the previous notation

$$-U = \frac{1}{2} - \frac{1}{2N}\sum_{i=1}^{N} E_i\,, \tag{3}$$

where N is the number of points in the lattice. In the limiting case of $N \to \infty$, U is known exactly

$$U = \frac{1 + [2\tanh^2(H) - 1]\,2K(k_1)/\pi}{\tanh(H)}\,, \tag{4}$$

where $H = 2Z$, $k_1 = 2\sinh(H)/\cosh^2(H)$, and $K(k_1) = F(\pi/2 \mid k_1)$ is the complete elliptic integral of the first kind [4,5]. This provides a way to compare the two algorithms.

Having defined the spin structure and the energy, the canonical scheme for changing spins can best be explained by the following piece of pseudo-code:

```
r = random number between 0 and 1
B = exp(-ΔE_i Z)
  if (r < B) then
    S_i = -S_i
  endif
```

where the temperature, $T = 1/Z$, is an input parameter of the program [6]. Executing the above procedure on every site in the lattice constitutes one iteration, or time step, of the algorithm.

The micro-canonical procedure is a bit more complicated. In addition to the spin lattice, we have a corresponding lattice of demons, D. Each element in D, D_i, is restricted by the condition that $D_i \in \{0, 1, 2, \ldots, D_{max}\}$.

D_i can be thought of as the kinetic energy conjugate to the ith point in the lattice. The most important property of the micro-canonical updating algorithm is that the energy at each point be conserved

$$E_i + 4D_i = \text{constant} \quad \Rightarrow \quad E_T = 4\sum_{i=1}^{N} D_i + \sum_{i=1}^{N} E_i \,, \tag{5}$$

where E_T is the total energy of the lattices. Eq. (5) illustrates the central difference between the two algorithms. Canonical sampling selects configurations based on their Boltzman weight, while micro-canonical sampling selects configurations that satisfy the total energy constraint of Eq. (5). As a consequence of Eq. (5)

$$\Delta E_i + 4\Delta D_i = 0 \,, \tag{6}$$

which gives some insight as to how to construct the updating algorithm. The spin flipping procedure used in our experiments was equivalent to the following code fragment:

```
D_i' = (4D_i – ΔE_i)/4
    if (0 ≤ D_i' ≤ D_max) then
        D_i = D_i'
        S_i = –S_i
    endif [7]
```

The factor of four comes about from the property—obtained from Eq. (1)—that $\Delta E_i \in \{-8, 4, 0, 4, 8\}$. A convenient way of running the program is to have each demon unit correspond to the smallest change in ΔE_i. For a detailed explanation of implementing this algorithm see ref. [8].

Eq. (6) obeys the principle of detailed balance [9], implying that S and D are governed by traditional thermodynamic principles. This provides an intuitive rational for believing that the micro-canonical algorithm works. As the system evolves, the demons become distributed according to their

Boltzman factors. The demon average will then be related to the temperature by

$$\frac{1}{N}\sum_{i=1}^{N} D_i = \langle D_i \rangle = \frac{\sum \gamma \exp(-4\gamma Z)}{\sum \exp(-4\gamma Z)}, \tag{7}$$

where the sums are carried out from $\gamma = 0$ to D_{max}. Given a particular state D, $\langle D_i \rangle$ can be calculated, and Eq. (5) can be numerically inverted to find the temperature. Thus, the temperature is an output rather than an input of the algorithm.

3. Running The Algorithms

Both algorithms were implemented with doubly periodic boundary conditions. A checkerboard updating procedure was used to avoid the result of [10]—that any algorithm which updates all spins simultaneously cannot simulate the Ising model. All the spins were initially set to +1. To compensate for the relaxation of the system from its starting state, some number of initial steps, t_i, need to be discarded. Berretti and Sokal [11] point out that t_i should not be larger than $\sim\frac{t_t}{4}$, t_t being the total number of iterations of the algorithm.

In the micro-canonical program the demons were initialized to either 2 or 0. An input parameter was used to set the ratio of the number of 2's to the number of 0's, depending upon the desired value of the total energy. In order to speed up the algorithm it was necessary to increase the thermal contact by "scrambling" the demons. This was done by offsetting the entire demon lattice by 3 positions in both the x and y directions after each iteration. The value of D_{max} needed to be set so that there were neither too few nor too many energy levels for the particular range of Z. The best

results seem to occur when D_{max} was allowed to vary dynamically so that the highest energy level contained ~0.1% of the demons.

A listing of the Fortran program used to implement these sampling techniques is given in Appendix B.

4. RESULTS

A survey of both algorithms using a 100x100 lattice (t_t = 1200 and t_i = 200), over the range $0 < Z < 1$, was conducted on a Stardent GK3000 mini-super-computer. The main output was the internal energy, U, which was calculated after each time step and then averaged for the whole run. These results (see Figs. 1a, b and Table I) illustrate the short term behavior of the algorithms. Both algorithms seem to follow the exact curve fairly well over the range $0.2 < Z < 0.7$. However, near the critical temperature, $Z_c = \frac{1}{2}\log(1+\sqrt{2}) = 0.44068...$, both produce results above the exact value (see Fig. 2). The kink in the internal energy that should occur at Z_c, seems to take place at $Z \approx 0.435$.

Comparing the relative accuracies of each method required much longer runs. These were carried out on a Cray 2 super-computer. The ratio of U to U_{exact} is shown in Fig. 3 (N = 100x100, t_t = 100,000 and t_i = 25,000). The numerical values are given in Table II. In general, one sees that the canonical is consistently much closer to the exact value. The micro-canonical appears to be correlated with Z. This correlation could be due to the fact that for finite N

$$\langle D_i \rangle = \frac{\sum \gamma \exp[-4\gamma Z - (4\gamma Z)^2/2CN] + O(1/N)}{\sum \exp[-4\gamma Z - (4\gamma Z)^2/2CN] + O(1/N)}, \tag{8}$$

where C is the specific heat (see ref. [3]). Runs of N = 40x40 and N = 200x200 (t_t = 100,000 and t_i = 25,000) were conducted for both algorithms. Figs. 4a

and 4b show U/U_{exact} for two different lattice sizes for the canonical and micro-canonical methods respectively. The canonical method changes little with lattice size, while the micro-canonical method exhibits increased variation with decreasing N. This variation could also be related to Eq. (8).

Another quantity of interest is the specific heat. In the canonical case, C can be calculated from the standard deviation of U, σ_U:

$$-C_{can} = Z^2 \frac{\partial U}{\partial Z} = Z^2(N\sigma_U{}^2) . \tag{9}$$

The derivation of Eq. (9) is given in Appendix A. Since both Z and U vary in the micro-canonical method, Eq. (9) cannot be used. That the micro-canonical method exhibits the same behavior as the canonical method can be shown using the following empirical equation

$$-C_{micro} = \frac{(N\sigma_Z{}^2)(N\sigma_U{}^2)}{K_m Z} , \tag{10}$$

where σ_Z is the standard deviation of Z, and K_m is a constant equal to one with units of energy. C_{can} and C_{micro} are plotted in Fig.5 (N = 100x100, t_t = 100,000, t_i = 25,000). Both agree with C_{exact} qualitatively, but fall short quantitatively. In the case of C_{can}, the shortfall is an indication that the successive configurations of S are correlated. This may also be true for C_{micro}, indicating that the correlation in the micro-canonical method is greater—a believable hypothesis since the algorithm is deterministic. C_{can}/C_{exact} and C_{micro}/C_{exact} are plotted in Fig. 6 for (N = 100x100, t_t = 1,000, t_i =250) and (N = 100x100, t_t = 100,000, t_i =25,000). Fig. 6 shows that there is a consistency in the difference between C_{can} and C_{micro}, and that both fall short of C_{exact} in the same way. This suggests that C_{micro} really does describe the specific heat. Fig. 6 also demonstrates that the variations due to different initial conditions are smoothed out as t_t is increased.

Finally, runs of N = 200x200, t_t = 1,000,000, t_i = 250,000 were conducted for both algorithms at $Z \approx 0.4$. No significant difference was found with the t_t = 100,000 data.

5. CONCLUSIONS

In this paper extensive surveys of the canonical and micro-canonical sampling methods were conducted over $0.2 < Z < 0.7$, $40x40 \leq N \leq 200x200$, and $1{,}000 \leq t_t \leq 1{,}000{,}000$. In each case U, Z, σ_U, and σ_Z, were examined. The main results are:

I) For short runs, t_t = 1,000, both algorithms give nearly identical results.

II) In longer runs, t_t = 100,000, it appears as though the more complex formulation of Z is required for the micro-canonical method to match the canonical method's accuracy.

III) The specific heat indicates that both algorithms suffer from successive configurations being correlated, with the micro-canonical algorithm suffering more.

For the researcher debating about which algorithm to use for his or her particular model, the main result of this work is that the two algorithms are equivalent enough in accuracy over $0.2 < Z < 0.7$, that the particular physics involved should be the determining factor. The canonical approach is simpler. One can get good results using a smaller lattice. Although, this performance advantage can be offset by the need for random numbers and the exponential function, depending upon the implementation. The main advantage of the micro-canonical approach is its temperature independence. This makes it ideal for studying systems where the temperature is either not uniform or evolving. Finally, a different set of

situations can be explored since the demons correspond to their own physical system—the kinetic energy of the lattice sites.

ACKNOWLEDGEMENTS

I would like to thank Prof. Alexandre Chorin for his guidance and support, Dr. M. Creutz, Dr. B. Alpert, and A. Qi for many helpful discussions, and the Department of Energy's Science and Engineering Research Semester for sponsoring me.

APPENDIX A

In this appendix the equation for the canonical specific heat is derived. From Eq.(9) it is apparent that

$$\frac{\partial U}{\partial Z} = N\sigma_U{}^2$$

is all that needs to be shown. First note:

$$\sigma_U{}^2 = \langle\, (\langle U\rangle - U)^2 \,\rangle = \langle U^2\rangle - \langle U\rangle^2 .$$

Let,

$$\mathcal{Z} = \sum_\mu \exp[-ZE_\mu]$$

and

$$\langle U\rangle = \frac{1}{\mathcal{Z}}\sum_\mu E_\mu \exp[-ZE_\mu]$$

be the partition function and the average internal energy of a thermodynamic system respectively [12]. $\langle U\rangle$ and $\langle U^2\rangle$ can be re-written in terms of partial derivatives

$$\langle U\rangle = -\frac{1}{\mathcal{Z}}\frac{\partial \mathcal{Z}}{\partial Z} \;\Rightarrow\; \langle U\rangle^2 = \frac{1}{\mathcal{Z}^2}\left(\frac{\partial \mathcal{Z}}{\partial Z}\right)^2$$

$$\langle U^2\rangle = \frac{1}{\mathcal{Z}}\sum_\mu E_\mu{}^2 \exp[-ZE_\mu] = \frac{1}{\mathcal{Z}}\frac{\partial^2 \mathcal{Z}}{\partial Z^2} .$$

Thus,

$$\sigma_U{}^2 = \frac{1}{\mathcal{Z}}\frac{\partial^2 \mathcal{Z}}{\partial Z^2} - \frac{1}{\mathcal{Z}^2}\left(\frac{\partial \mathcal{Z}}{\partial Z}\right)^2 = \frac{\partial}{\partial Z}\left(\frac{1}{\mathcal{Z}}\frac{\partial \mathcal{Z}}{\partial Z}\right).$$

Substituting in for the definition of $\langle U\rangle$ gives

$$\sigma_U{}^2 = \frac{\partial\langle U\rangle}{\partial Z} = \left(\frac{\partial U}{\partial Z}\right)_{U\,=\,\langle U\rangle} .$$

In the finite 2D Ising model [13]

$$U \rightarrow -\frac{\partial}{\partial Z}\left(\frac{1}{N}\log(\mathcal{Z}_N)\right),$$

which has the necessary factor of N, resulting in

$$\frac{1}{N}\frac{\partial U}{\partial Z} = \sigma_U{}^2$$

REFERENCES

1. K. BINDER, *Applications of the Monte Carlo Method in Statistical Physics* (Springer-Verlag, Berlin, 1987), p. 5.
2. M. CREUTZ, *Annals of Physics* **167**, 62 (1986).
3. G. BHANOT, M. CREUTZ, AND H. NEUBERGER, *Nuclear Physics* B **235** [FS11], 417 (1984).
4. L. ONSAGER, *Physical Review* **65**, 118 (1944).
5. C. J. THOMPSON, *Mathematical Statistical Mechanics* (MacMillan, New York, 1972), pp. 131-135.
6. K. BINDER, *Applications of the Monte Carlo Method in Statistical Physics* (Springer-Verlag, Berlin, 1987), p. 10.
7. M. CREUTZ, *Physical Review Letters* **50**, 1411 (1983).
8. H. GOULD AND J. TOBOCHNIK, *An Introduction to Computer Simulation Methods: Applications to Physical Systems* (Addison-Wesley, 1988), Part II, pp. 501-525.
9. C. KITTEL AND H. KROEMER, *Thermal Physics* (W. H. Freeman and Co., New York, 1980), 2nd edition, pp. 407.
10. G. VICHNIAC, *Physica D* **10**, 96 (1984).
11. A. BERRETTI, A. D. SOKAL, *J. of Statistical Physics* **40**, 483 (1985).
12. C. KITTEL AND H. KROEMER, *Thermal Physics* (W. H. Freeman and Co., New York, 1980), 2nd edition, pp. 83-84.
13. A. CHORIN, *Communications in Mathematical Physics* **99**, 501 (1985).

TABLE I

Some numerical values of the internal energy calculated from the canonical and micro-canonical algorithms of the 2D Ising model. Programs were run for 1200 iterations on a 100x100 lattice. The first 200 iterations were discarded. "–" indicates that the algorithm did not yield consistent results. D_{max} is the largest a demon could become for the given micro-canonical run.

	U			
Z	Exact	Can	Micro	D_{max}
0	0	–	–	–
0.1064	0.2169	0.2497	0.2174	14
0.2040	0.4381	0.4422	0.4381	8
0.3091	0.7339	0.7330	0.7324	5
0.4033	1.1240	1.1223	1.1215	4
0.4370	1.3653	1.3916	1.3987	4
0.4407	1.4151	1.4113	1.4114	4
0.5142	1.7829	1.7801	1.7808	3
0.6733	1.9540	1.9522	1.9455	2
1	1.9972	1.9955	–	–

TABLE II

Numerical values used to generate Fig. 3 and Fig. 5.
Data obtained from runs using N = 100x100, t_t = 100,000, and t_i = 25,000.

Canonical				Micro-Canonical				
Z	U	$N\sigma_U^2$	U/U_{ex}	Z	$N\sigma_Z^2$	U	$N\sigma_U^2$	U/U_{ex}
0.20	0.42827	1.34756	1.00010	0.21099	0.00700	0.45479	1.16199	0.99875
0.25	0.55728	1.82468	1.00001	0.26849	0.01626	0.60867	1.45837	0.99914
0.30	0.70450	2.39080	1.00000	0.32558	0.03862	0.78891	1.70429	0.99894
0.35	0.87990	3.23888	1.00011	0.37865	0.08173	0.99968	1.95769	0.99925
0.40	1.10587	4.87621	0.99981	0.42220	0.14297	1.23924	2.23533	0.99835
0.45	1.51312	7.47434	1.00005	0.44118	0.19915	1.42264	2.61784	0.99961
0.50	1.74558	2.58469	1.00001	0.46171	0.18690	1.59006	1.82139	0.99934
0.55	1.85104	1.29418	0.99995	0.49603	0.18469	1.73236	1.20608	0.99922
0.60	1.90906	0.73575	0.99999	0.52703	0.18775	1.81125	0.86258	1.00017
0.65	1.94306	0.43015	0.99999	0.55614	0.18912	1.86026	0.65435	1.00007
0.70	1.96380	0.26043	1.00001	0.60289	0.21157	1.91239	0.47006	1.00043

Figure Texts

Fig. 1. Internal energy vs. inverse temperature for (a) canonical and (b) micro-canonical approaches (N = 100x100, t_t = 1200, t_i = 200).

Fig. 2. Behavior of the the internal energy near the critical temperature (N = 100x100, t_t = 1200, t_i = 200).

Fig. 3. Ratio of the exact to the computed values of the internal energy (N = 100x100, t_t = 100,000, t_i = 25,000).

Fig. 4. Effect of changing the lattice size, N, on (a) canonical and (b) micro-canonical approaches (t_t = 100,000, t_i = 25,000).

Fig. 5. Specific heat (C_{can} and C_{micro}) vs. inverse temperature (N = 100x100, t_t = 100,000, t_i = 25,000).

Fig. 6. Effect of increasing the total number of time steps, t_t, on C_{can}/C_{exact} and C_{micro}/C_{exact} (N = 100x100).

Fig. 1a.

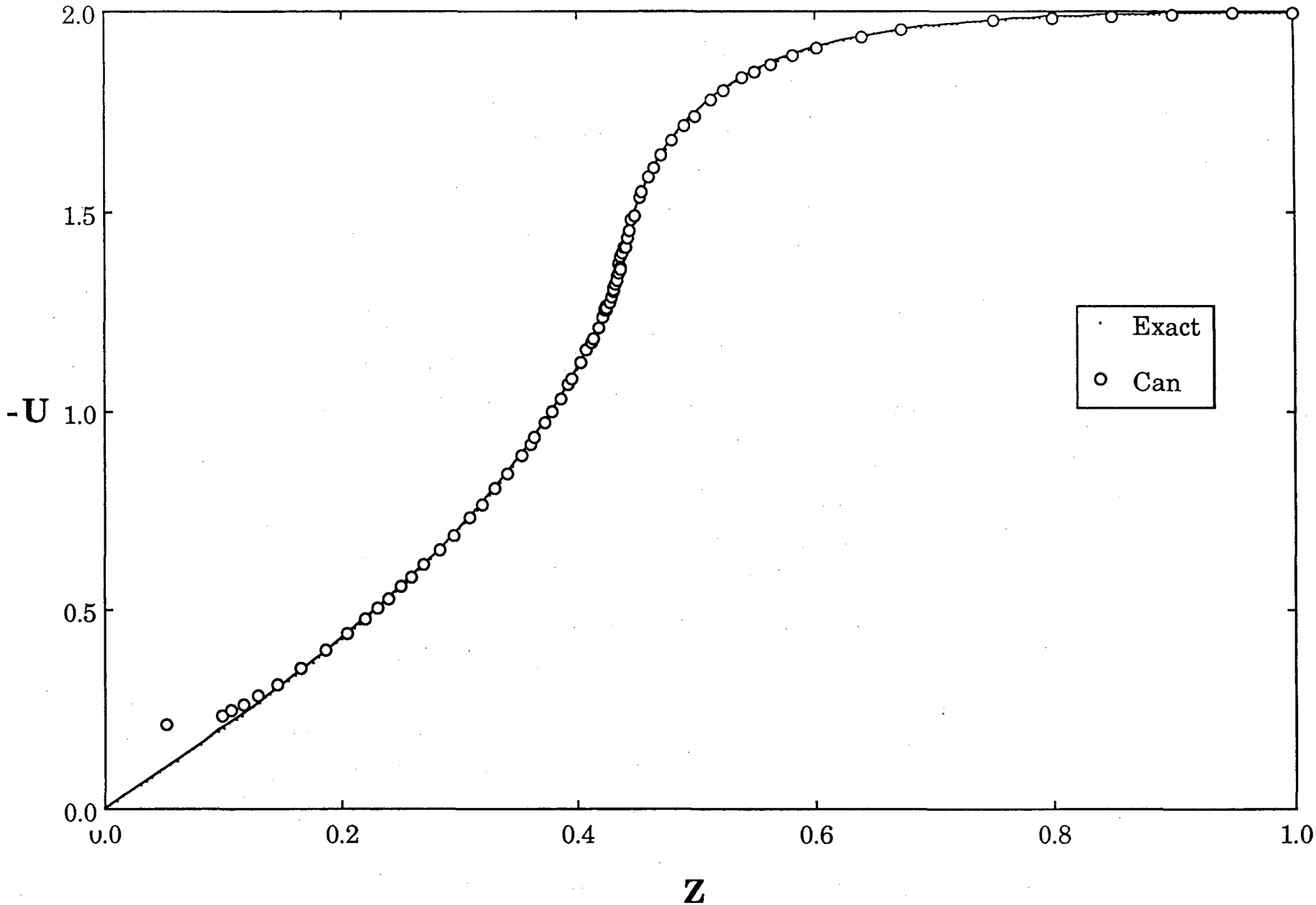


Fig. 1b.

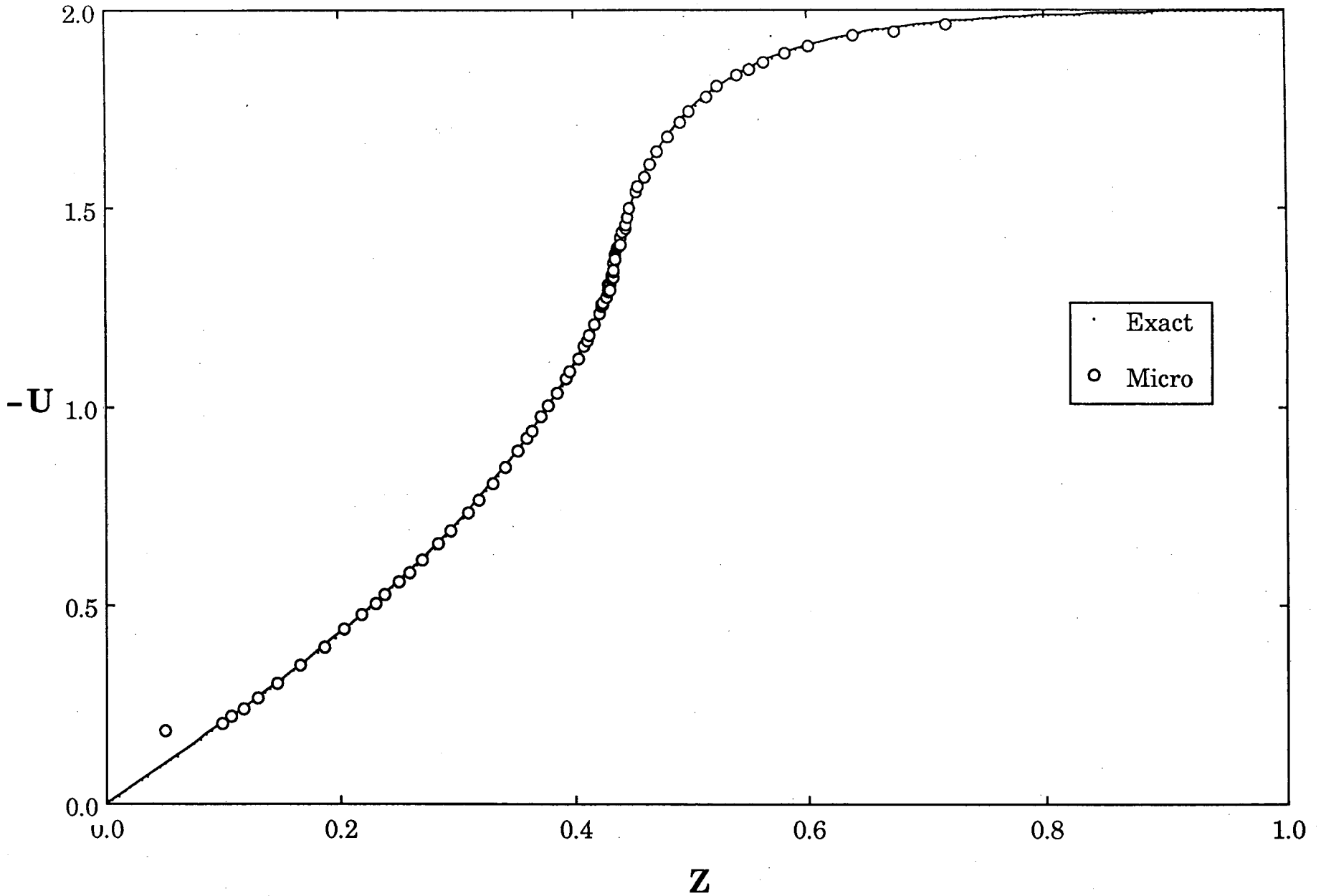


Fig. 2.

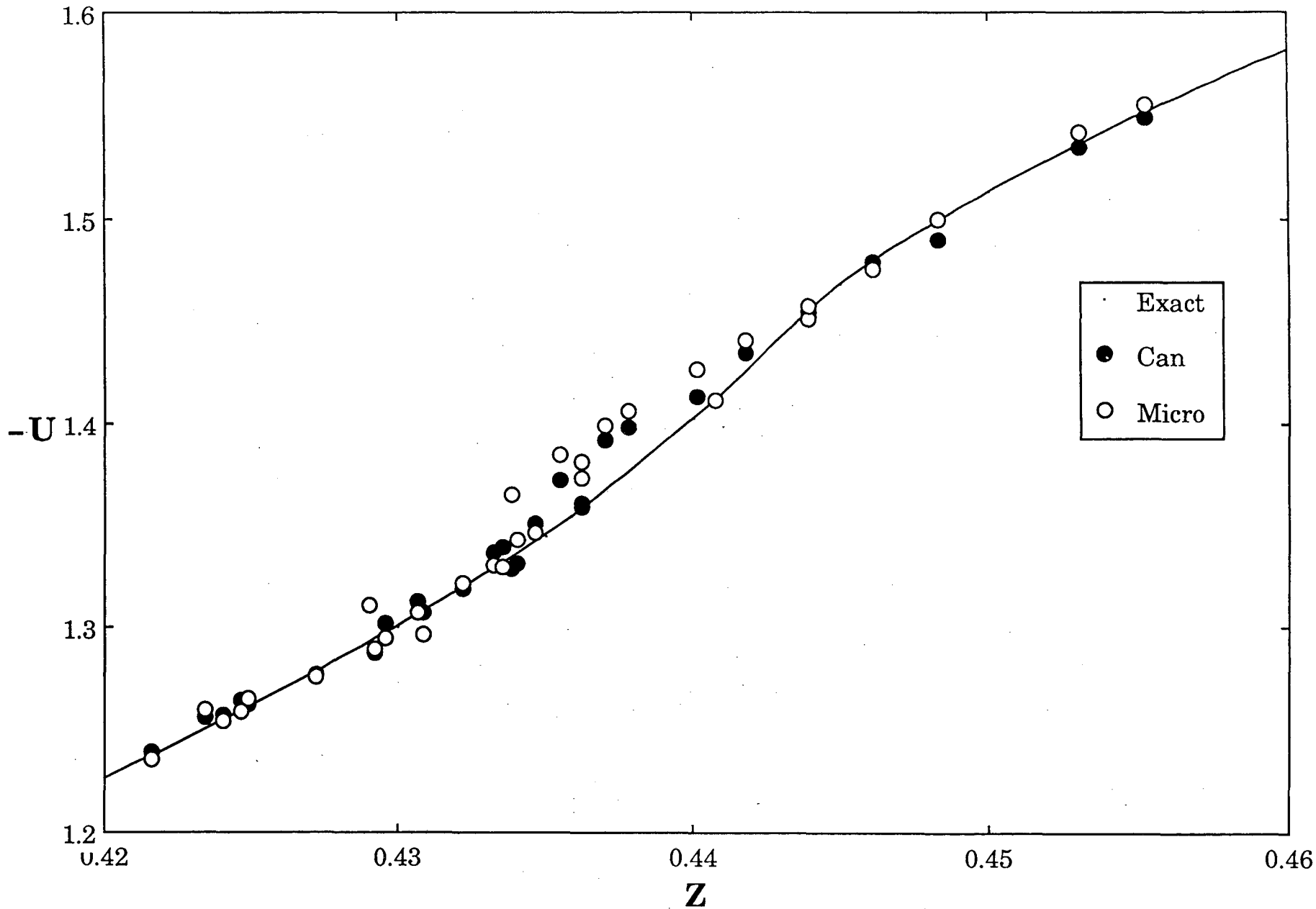


Fig. 3.

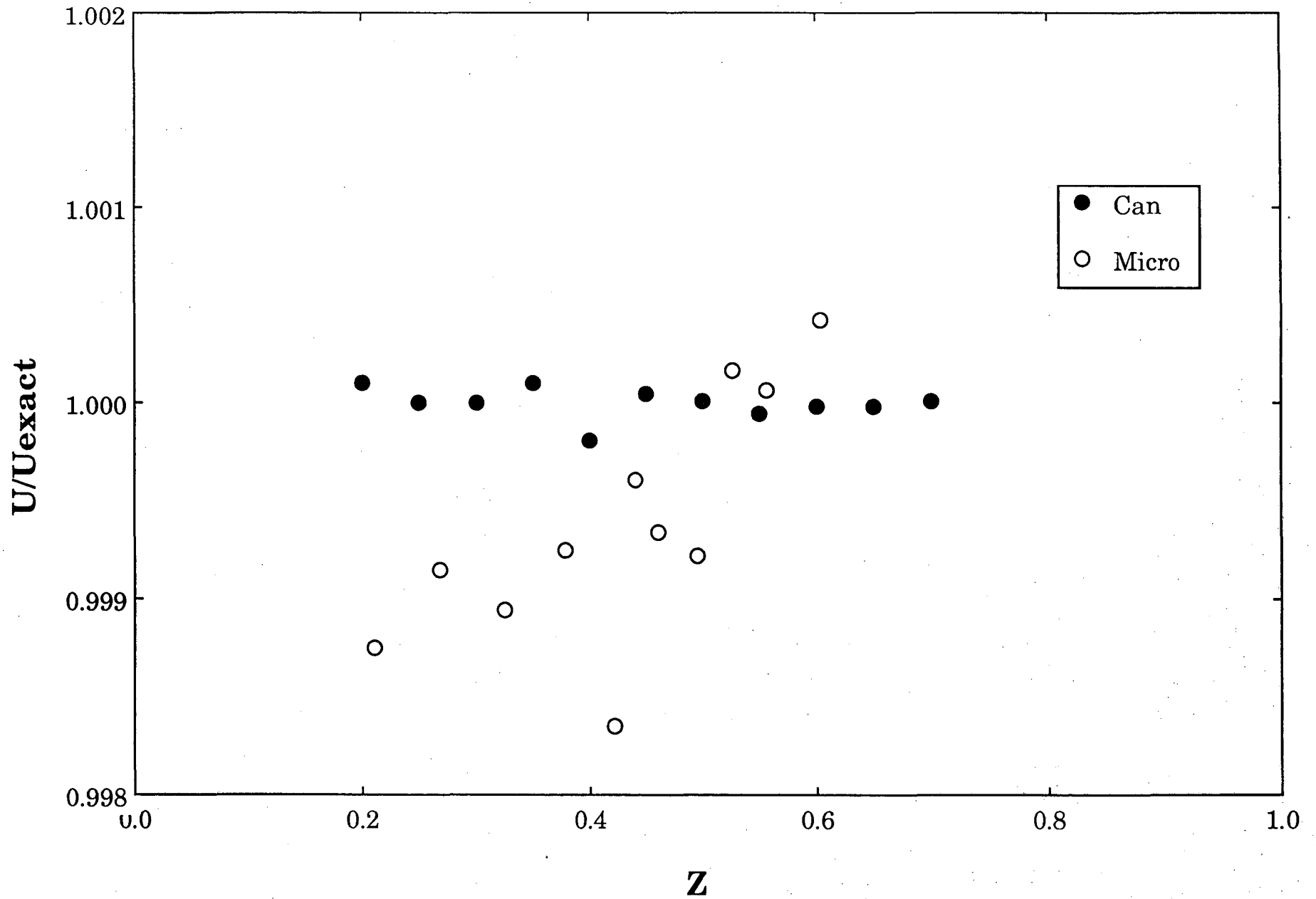


Fig. 4a.

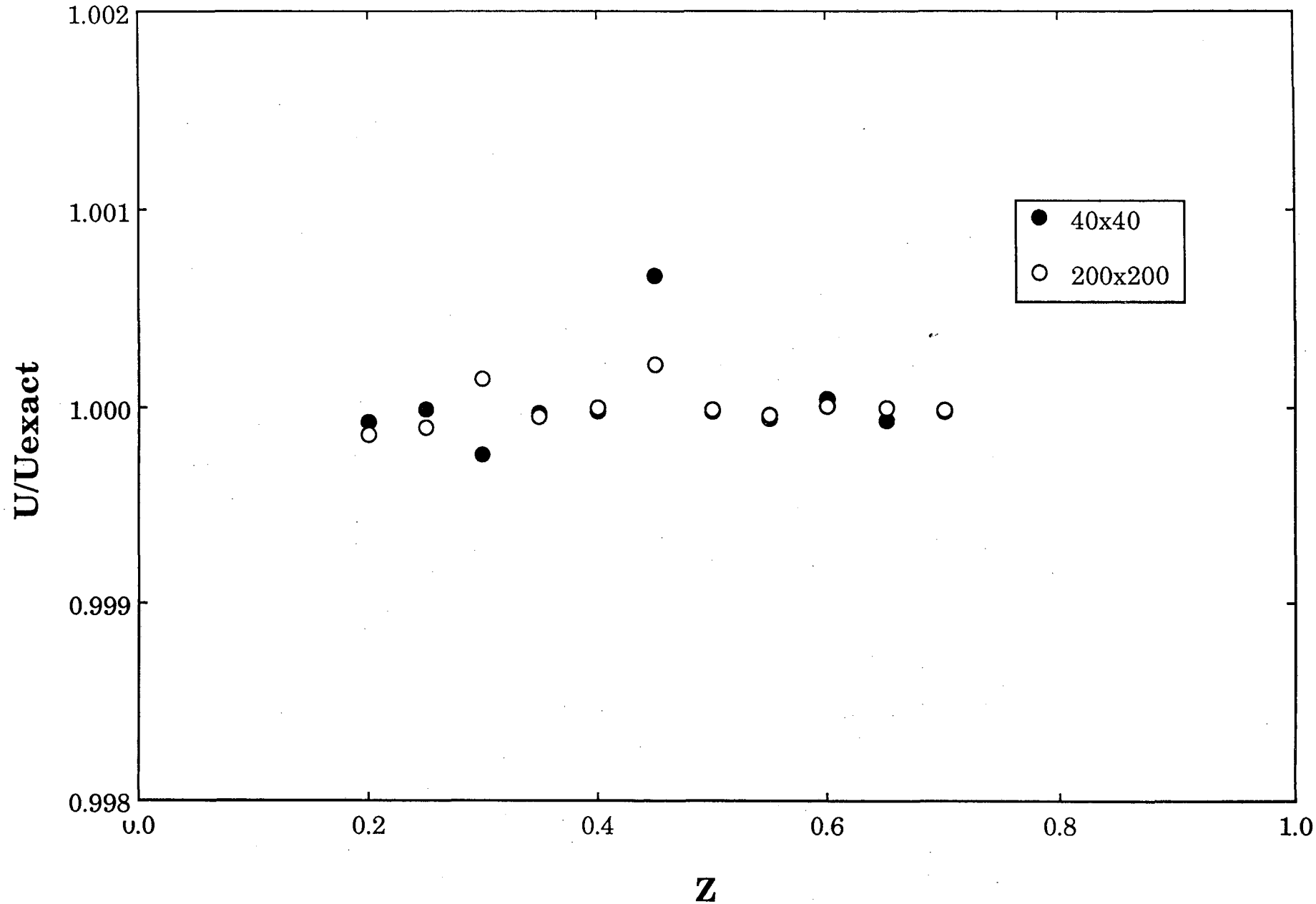

1.002
1.001
1.000
0.999
0.998
U/Uexact
0.0
0.2
0.4
0.6
0.8
1.0
Z
40x40
200x200

Fig. 4b.

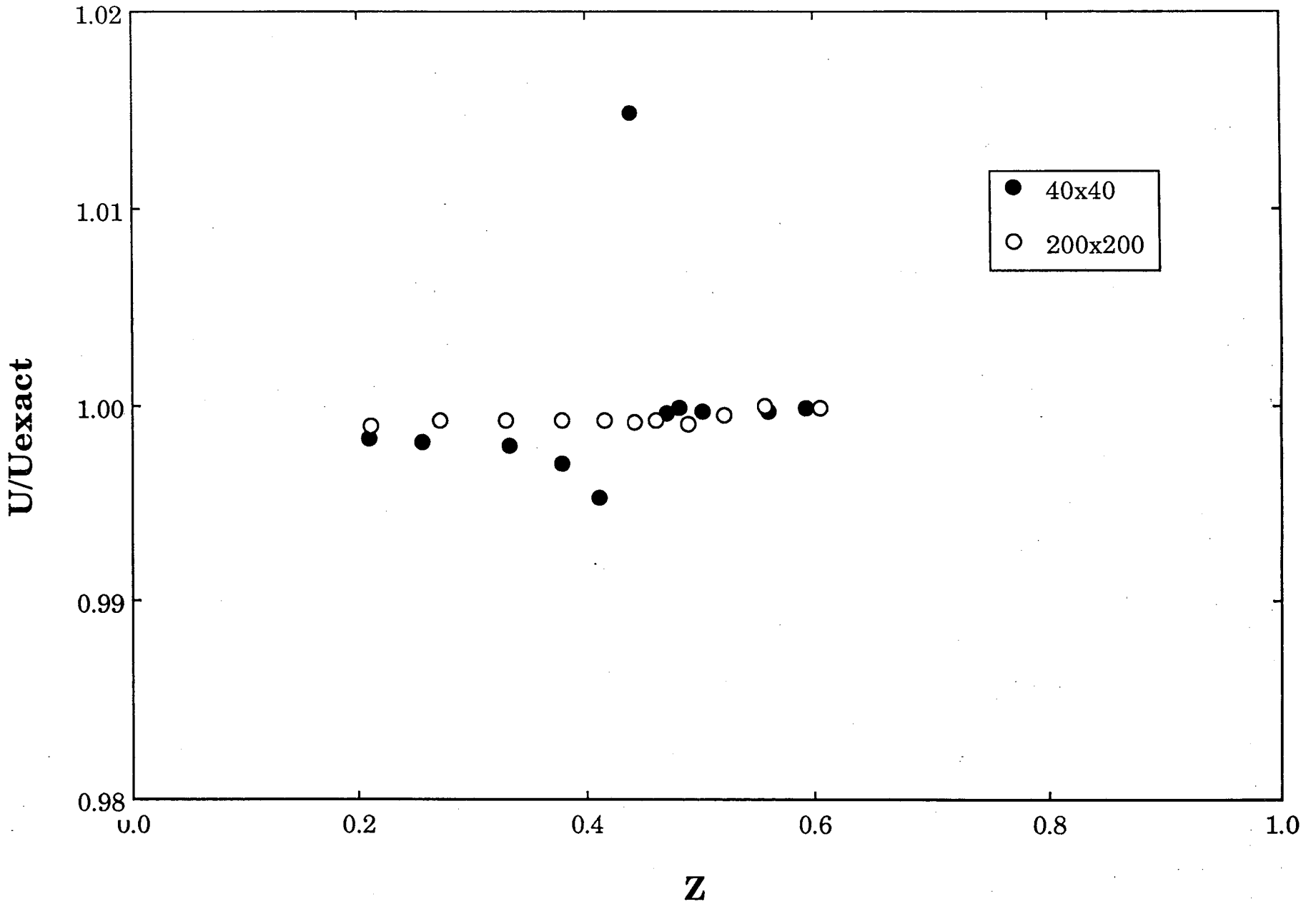


Fig. 5.

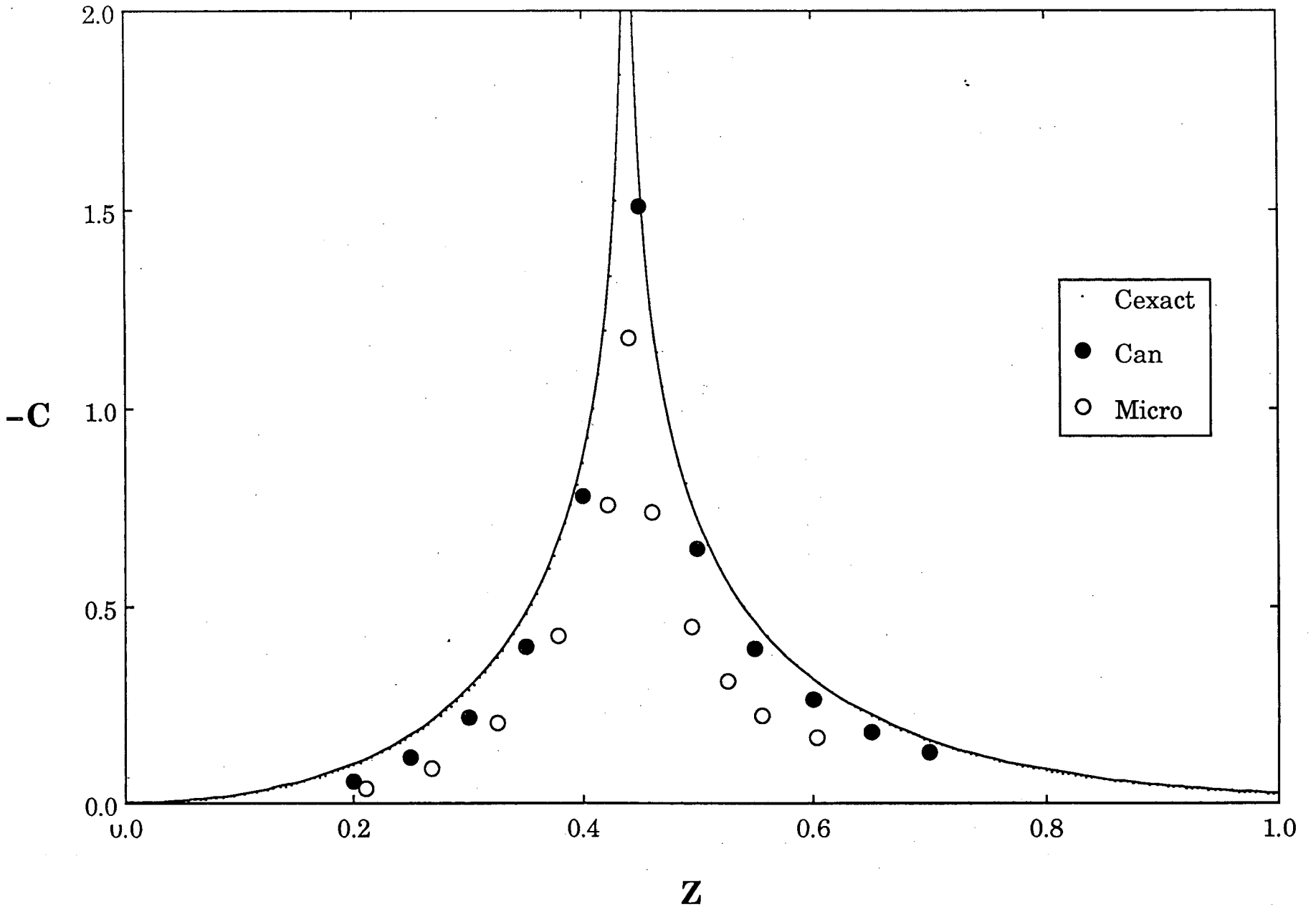
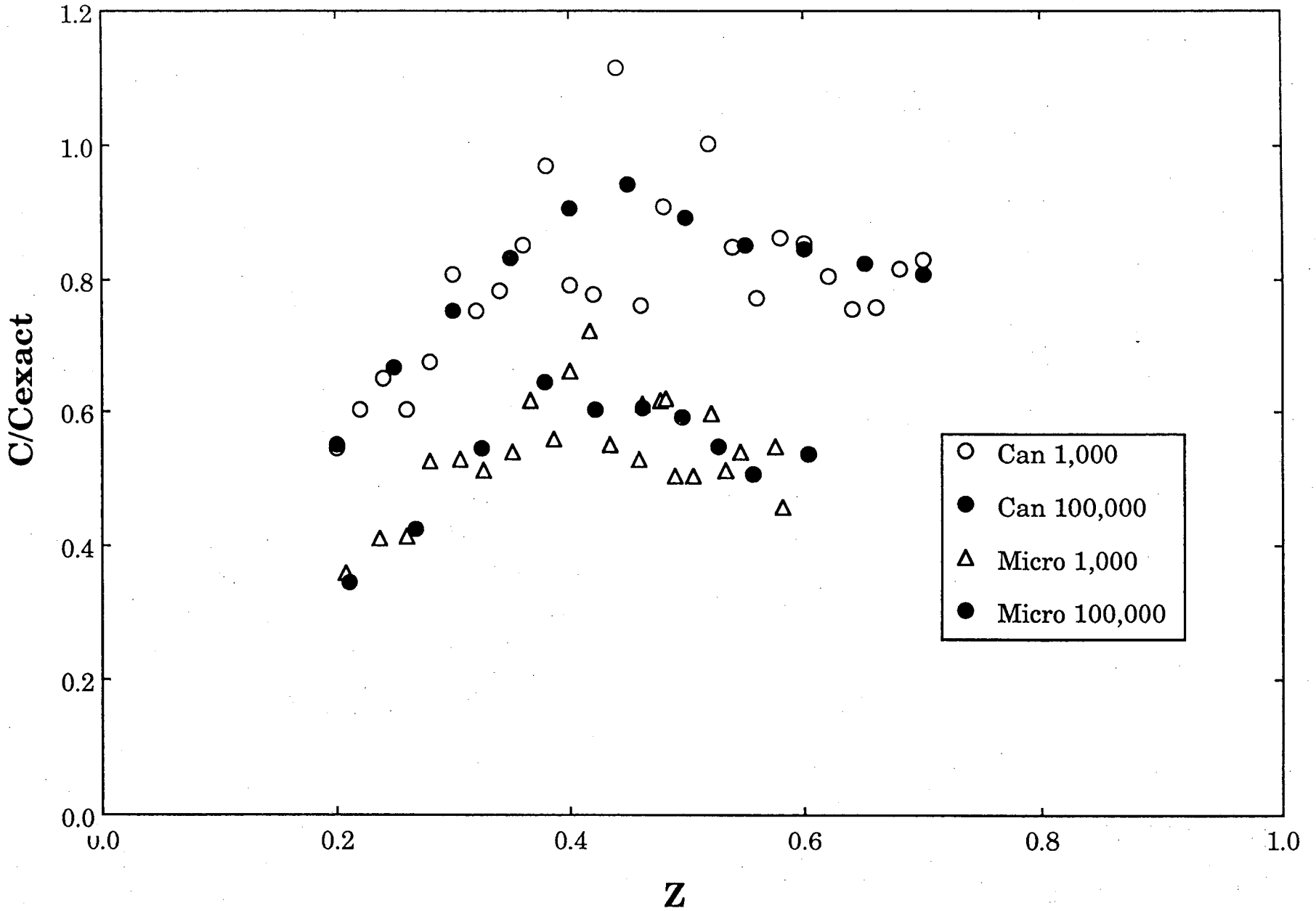


Fig. 6.


1.2
1.0
0.8
0.6
0.4
0.2
0.0
C/Cexact
0.0
0.2
0.4
0.6
0.8
1.0
Z
Can 1,000
Can 100,000
Micro 1,000
Micro 100,000

APPENDIX B

Listing of the Fortran program used in these experiments. Note that the first section is the common block, which is inserted by the "include" command at the beginning of each subroutine.

```
c       ******************* BEGIN COMMON BLOCK **************

c       calculation variables
c       302*302 = 91204
        integer gspin1(302,302), gdemarr1(302,302)
        integer gspin2(302,302), gdemarr2(302,302)
        integer gmap(91204,6)

c       gspin1 = SPIN array 1; gdemarr1 = DEMon ARRay 1
c       gspin1 = SPIN array 2; gdemarr1 = DEMon ARRay 2
c       gmap = MAP array

        common gspin1, gdemarr1
        common gspin2, gdemarr2
        common gmap

c       calculation parameters
        logical gdovarb
        integer gmethod, gdemoff
        integer gmaxbins, gminbins, gmaxdem
        integer gbinht, gdemonht
        real gvarbwt, gspinwt, gdemonwt

c       gdovarb = DO VARiable Bins
c       gmethod = sampling METHOD; gdemoff = DEMon array OFFset
c       gmaxbins = MAXimum demon BIN; gminbins = MINimum demon BIN;
c         gmaxdem = MAXimum DEMon value
c       gbinht = demon BIN HeighT; gdemonht = DEMON value HeighT
c       gvarbwt = VARiable Bin WeighT; gspinwt = SPIN WeighT;
c         gdemonwt = DEMON WeighT


        common gdovarb
        common gmethod, gdemoff
        common gmaxbins, gminbins, gmaxdem
        common gbinht, gdemonht
        common gvarbwt, gspinwt, gdemonwt


c       canonical globals
        real gzinput, gboltzar(5)

c       gzinput = Z Input; gboltzar = BOLTZman ARray

        common gzinput, gboltzar


c       calculation outputs
        integer glastbin
```

```
      real gavgbin, gavgham, gz, gu

c     glastbin = LAST demon BIN
c     gavgbin = AVeraGe demon BIN; gavgham = AVeraGe spin HAMiltonian;
c       gz = Z (inverse temperature); gu = U (internal energy)

      common glastbin
      common gavgbin, gavgham, gz, gu


c     running outputs
      integer gstep
      real gztot, gutot
      real gzsqrd, gusqrd

c     gstep = current time STEP
c     gztot = Z TOTal; gutot = U TOTal
c     gzsqrd = total Z SQuaReD; gusqrd = total U SQuaReD

      common gstep
      common gztot, gutot
      common gzsqrd, gusqrd

c     running paramters
      integer gxsize, gysize, gtotal, gsteps
      integer gavgit, gfullu

c     gxsize = X SIZE; gysize = Y SIZE;
c       gtotal = TOTAL array elements; gsteps = total time STEPS
c     gavgit = time steps before calling AVeraGe IT;
c       gfullu = FULL output file Unit

      common gxsize, gysize, gtotal, gsteps
      common gavgit, gfullu

      character*8 gfullf
c     gfullf = FULL output Filename
c     Cray insists that all string variables be placed
c     in a separate common block.
      common /string/gfullf

c     ******************* END COMMON BLOCK **************

    program ising
c     load common block
      include 'vectb.f'

c     global/common block variables are denoted by a "g"
c     Declare local variables.
c     nstep is the size of the lattice, zstep is
c     the temperature, and zscale converts
c     zstep into a usable real number.

      integer nstep,zstep
      real zscale
      character*16 title1,title2

c      call dropfile(0)
```

```
c       Set defualt values.

        call setdefs
        zscale = 0.001

c       Set up output file.
c       Since we are using status='old', a file
c       called gfile (see defaults) must already exist.

        title1 = 'n     avgz  sigz  uex   '
        title2 = 'avgu  sigu  cex'
        open(unit=gfullu,file=gfullf,status='old')
        write(gfullu,*) title1,title2

c       This is where the user can change values
c       e.g. time step, sampling method, ...
c       Vary the constants in the zstep and nstesp
c       do loops to change the temperature and
c       lattice size.  These loops are there to make
c       it convenient to do surveys over a range of
c       temperatures and lattice sizes.

        gsteps = 40
        gmethod = 1
        gdovarb = .true.
        gavgit = gsteps/4
        do 90 zstep = 200,200,100
          gzinput = zscale * zstep
          if (gmethod .eq. 1) then

c           Approximate imperical formula that
c           sets the initial demon lattice so that the
c           equilibrium temperature will be approximately
c           equal to gzinput.  This is only relavent
c           in the micro-canonical (gmethod = 1).

            gdemonwt =  ((1.0/gzinput) - 1.0)/7.0

            gdemoff = 3
          endif

          do 80 nstep = 40,40,20

c           Although have seperate variables for the
c           X and Y array limits, this is only there
c           if one day someone wants to do rectagular
c           arrays.

            gxsize = nstep
            gysize = gxsize
            gtotal = gxsize * gysize
            print*,'z = ',gzinput,' n = ',gxsize
            call initit

c           Begin main loop.

            do 70 gstep = 1, gsteps
```

```
c             Choose between micro-canonical
c             and canonical approaches

              if (gmethod .eq. 1) then
                call calcmic
                call evaluate
                if (gdemoff .gt. 0) call randemon
              elseif (gmethod .eq. 2) then
                call calccan
                call evaluate
              endif
              if (gstep .ge. gavgit) call avgit
   70       continue
          call stats
   80     continue
   90   continue
        close(gfullu)
      end


c     Converts bins to demons.
      function bindem(bins)
        integer bins
        bindem = 4 * (bins - 1)
      return
      end


c     Converts demons to bins.
      function dembin(demon)
        integer demon
        dembin = (demon/4) + 1
      return
      end


c     Sets the default values of many of
c     the variables in the common block.
      subroutine setdefs
c       load common block
        include 'vectb.f'

        gmethod = 2
        gzinput = 0.4
        gdemoff = 0
        gdovarb = .true.
        gvarbwt = 0.001
        gmaxbins = 5
        gminbins = 3
        gmaxdem = bindem(gmaxbins)
        gspinwt = 0.0
        gdemonwt = 0.0
        gbinht = 3
        gdemonht = bindem(gbinht)

        gxsize = 100
        gysize = gxsize
```

```
        gtotal = gxsize * gysize
        gsteps = 100000
        gavgit = 200
        gfullf = 'fulld'
        gfullu = 21

      return
      end


c     INITIALIZE ARRAYS
c     Initialize the lattice in accordance with
c     the values set at the beginning of the main
c     loop.  The function rand(0) returns a random
c     real number between 0.0 and 1.0 with uniform
c     distribution.

      subroutine initit
c       load common block
        include 'vectb.f'
        integer i,j,k,l,s,d,ip,im,jp,jm


c       INIT BOLTZMAN TABLE
c       No need to call exp() all the time
c       since there are only five different
c       values in the canonical method, so we put
c       them in a lookup table.

        do 10 i = 1, 5
          gboltzar(i) = exp(4*(3-i)*gzinput)
   10   continue


c       INITMAP
c       In order to use checkerboard updating need
c       to create a list of x and y lattice locations
c       so that one can simply step through the list
c       and hit the right points.  NOTE, this requires
c       that gxsize and gysize be even.
c       Stepping through the first half of the map
c       gives the first color, the second half
c       gives the second color.

        k = 0
        l = gtotal/2
        do 30 i = 1, gxsize - 1, 2
          do 20 j = 1, gysize - 1, 2
            k = k + 1
            gmap(k,1) = i
            gmap(k,2) = j
            k = k + 1
            gmap(k,1) = i + 1
            gmap(k,2) = j + 1
            l = l + 1
            gmap(l,1) = i
            gmap(l,2) = j + 1
```

```
            l = l + 1
            gmap(l,1) = i + 1
            gmap(l,2) = j
   20     continue
   30   continue


c       INIT SPINS
c       Sets initial configuration of the spin lattice.
c       Lowest energy is when spins are
c       either all up, 1, or all down, -1,
c       which corresponds to gspinwt of 1 and 0
c       respectively. The highest energy state
c       corresponds to gspinwt = 0.5.

        do 40 k = 1, gtotal
          i = gmap(k,1)
          j = gmap(k,2)

c         weights spins according to gspinwt
          s = -1
          if (rand(0) .lt. gspinwt) s = 1
          gspin1(i,j) = s
   40   continue

c       INIT DEMONS
c       Sets initial configuration of demon lattice.
c       All demons are set to either 0 or gdemonht in
c       praportion to gdemonwt.  Thus the highest
c       energy is gdemonwt = 1.0

        do 50 k = 1, gtotal
          i = gmap(k,1)
          j = gmap(k,2)

c         weights demons according to gdemonwt
          d = 0
          if (rand(0) .lt. gdemonwt) d = gdemonht
          gdemarr1(i,j) = d
   50   continue


c       INIT COPIES
c       In order to vectorize, need to have copies
c       of the arrays.  So, we need to copy the
c       initial values of the arrays to their
c       corresponding copies.
        do 60 k = 1, gtotal
          i = gmap(k,1)
          j = gmap(k,2)
          gspin2(i,j) = gspin1(i,j)
          gdemarr2(i,j) = gdemarr1(i,j)
   60   continue

c       PERIODIC BOUNDARY CONDITIONS
c       Called "torroidal" or "doubly periodic".
c       For a given lattice size these conditions
c       are constant and can be calculated in
```

```
c       advance and put into the map, so that
c       they only need be looked up.  In theory
c       it is the fastest approach.
        do 70 k = 1, gtotal
          i = gmap(k,1)
          j = gmap(k,2)

          ip=i+1
          im=i-1
          jp=j+1
          jm=j-1
          if (im .lt. 1) im = gxsize
          if (ip .gt. gxsize) ip = 1
          if (jm .lt. 1) jm = gysize
          if (jp .gt. gysize) jp = 1
          gmap(k,3) = ip
          gmap(k,4) = im
          gmap(k,5) = jp
          gmap(k,6) = jm
   70   continue

      return
      end


c     CYCLE DEMONS
c     To speed up the relaxation time of
c     the micro-canonical approach, the demons
c     are cycled so as to increase the rate at
c     which energy is transferred around the lattice.
c     Without it, oscillations between the spin
c     lattice and the demon lattice can occur.

      subroutine randemon
c       load common block
        include 'vectb.f'
        integer i,j,xoffset,yoffset,xmoveto,ymoveto

c       Get x and y positions from map, no implicit
c       need to, but helps vectorization.
c       Offset demon and move into the copy, then copy
c       back into demon array.  In this instance the
c       copy, gdemarr2, just acts as a convenient
c       storage space.

        xoffset = gdemoff
        yoffset = gdemoff
        if (gdemoff .gt. 0) then
          do 10 k = 1, gtotal
            i = gmap(k,1)
            j = gmap(k,2)
            xmoveto = i + xoffset
            ymoveto = j + yoffset
            if (xmoveto .gt. gxsize) xmoveto = xmoveto - gxsize
            if (ymoveto .gt. gysize) ymoveto = ymoveto - gysize
            gdemarr2(xmoveto,ymoveto) = gdemarr1(i,j)
   10     continue
          do 20 k = 1, gtotal
```

```
          i = gmap(k,1)
          j = gmap(k,2)
          gdemarr1(i,j) = gdemarr2(i,j)
20      continue
      endif
    return
    end


c     MICRO-CANONICAL
c     Main subroutine for micro-canonical.
c     Jump through quite a few hoops to get
c     vectorization. 1) Update gspin1
c     and gdemarr1 from gspin2 and gdemarr2 for
c     first color of the checkerboard. 2) Copy
c     spin1 and gdemarr1 back into gspin2
c     and gdemarr2. 3) Repeat calculations for
c     the second color of the checkerboard.
c     WARNING: Updating all spins at once causes the
c     algorithm to fail, which is why we use the
c     checkerboard updating scheme.

      subroutine calcmic
c       load common block
        include 'vectb.f'
        integer i,j,k,l,lo,hi,isum
        integer newdemon,delh,demsum,hamsum
        logical test

c       glastbin is used to vary bins.
        glastbin = 0
        demsum = 0
        hamsum = 0

        do 30 l = 1,2

          if (l .eq. 1) then
            lo = 1
            hi = gtotal/2
          elseif (l .eq. 2) then
            lo = 1 + gtotal/2
            hi = gtotal
          endif

          do 10 k = lo,hi
            i = gmap(k,1)
            j = gmap(k,2)
            isum = gspin2(gmap(k,3),j) + gspin2(gmap(k,4),j)
     a       + gspin2(i,gmap(k,5)) + gspin2(i,gmap(k,6))
            delh = 2 * gspin2(i,j) * isum
            newdemon = gdemarr2(i,j) - delh
            test = ((newdemon.ge.0).and.(newdemon.le.gmaxdem))
            if (test) gdemarr1(i,j) = newdemon
            if (test) gspin1(i,j) = -gspin2(i,j)
            hamsum = hamsum + (gspin2(i,j) * isum)
10        continue

          do 20 k = lo,hi
```

```
          i = gmap(k,1)
          j = gmap(k,2)
          gspin2(i,j) = gspin1(i,j)
          gdemarr2(i,j) = gdemarr1(i,j)
          if (gdemarr1(i,j) .eq. gmaxdem) glastbin = glastbin + 1
          demsum = demsum + gdemarr1(i,j)
20      continue

30    continue
      gavgbin = demsum/(4.0*gtotal)
      gavgham = hamsum/(1.0*gtotal)
    return
    end


c     CANONICAL
c     Main subroutine for canonical.
c     Jump through quite a few hoops to get
c     vectorization. 1) Update gspin1
c     from gspin2 for
c     first color of the checkerboard. 2) Copy
c     spin1 back into gspin2.
c     3) Repeat calculations for
c     the second color of the checkerboard.
c     WARNING: Updating all spins at once causes the
c     algorithm to fail, which is why we use the
c     checkerboard updating scheme.

      subroutine calccan
c       load common block
        include 'vectb.f'
        integer k,l,i,j,isum,lo,hi
        integer delh, aboltz, hamsum
        logical test

        hamsum = 0
        do 30 l = 1, 2
          if (l .eq. 1) then
            lo = 1
            hi = gtotal/2
          elseif (l .eq. 2) then
            lo = 1 + gtotal/2
            hi = gtotal
          endif
          do 10 k = lo,hi
            i = gmap(k,1)
            j = gmap(k,2)
            isum = gspin2(gmap(k,3),j) + gspin2(gmap(k,4),j)
     a       + gspin2(i,gmap(k,5)) + gspin2(i,gmap(k,6))
            delh = 2 * gspin2(i,j) * isum
            aboltz = 3 + (delh/4)
            test = rand(0) .lt. gboltzar(aboltz)
            if (test) gspin1(i,j) = -gspin2(i,j)
            hamsum = hamsum + (gspin2(i,j) * isum)
10        continue

          do 20 k = lo,hi
            i = gmap(k,1)
```

```
            j = gmap(k,2)
            gspin2(i,j) = gspin1(i,j)
20        continue
30      continue
        gavgham = hamsum/(1.0*gtotal)
      return
      end


c     Takes values calculated from the main
c     loop and acts on them.  Finds the energy
c     and calculates the proper temperature
c     depending upon which sampling method
c     is being used.  Also, the variable
c     bin algorithm is implemented here.

      subroutine evaluate
c       load common block
        include 'vectb.f'
        logical lolim, hilim, lobin, hibin
        real eps

        gu = (gavgham/2.0)

        if (gmethod .eq. 1) then
          gz = findz(gavgbin,gmaxbins)
        elseif (gmethod .eq. 2) then
          gz = gzinput
        endif

c       Want to make sure gmaxdem is neither too
c       large nor too small.  glastbin tells us
c       how many demons are in the highest energy
c       level.  The number we are aiming for is
c       gvarbwt * gtotal.  NOTE: that there are
c       bounds below (gminbins), and above (32).
c       NOTE2: that there is a region, 0.5 to 1.5,
c       times the target, in which no change
c       occurs.  This helps prevent oscillations.

        if ((gdovarb).and.(gmethod.eq.1)) then
          eps = gvarbwt * gtotal
          lolim = gmaxbins .gt. gminbins
          hilim = gmaxbins .lt. 32
          lobin = glastbin .lt. (0.5 * eps)
          hibin = glastbin .gt. (1.5 * eps)
          if ((lolim) .and. (lobin)) then
            gmaxbins = gmaxbins - 1
            gmaxdem = bindem(gmaxbins)
          elseif ((hilim) .and. (hibin)) then
            gmaxbins = gmaxbins + 1
            gmaxdem = bindem(gmaxbins)
          endif
        endif
      return
      end
```

```
c     Keeps track of values needed to
c     calculate the mean and standard
c     deviation of Z and U.

      subroutine avgit
c       load common block
        include 'vectb.f'
        character tab
        tab = '    '
c       initialize avg counters first time through
        if (gstep .le. gavgit) then
          gutot = 0.0
          gusqrd = 0.0
          gztot = 0.0
          gzsqrd = 0.0
        elseif (gstep .gt. gavgit) then
          gutot = gutot + gu
          gusqrd = gusqrd + ((gu)**2)
          if (gmethod .eq. 1) then
            gztot = gztot + gz
            gzsqrd = gzsqrd + ((gz)**2)
          endif
        endif
      return
      end


c     Calculates mean and standard deviation
c     of Z and U from the values calculated
c     in the subroutine avgit.  Also computes
c     the exact values of U and the specific
c     heat, C, from Z; writes these values
c     into the output file.

      subroutine stats
c       load common block
        include 'vectb.f'
        integer effsteps
        real totz, totu
        real avgz,sigz,uex,avgu,sigu,cex
        character t
        t = '      '

        effsteps = gsteps - gavgit
        avgz = gztot/effsteps
        avgu = gutot/effsteps
        totz = gzsqrd - ((gztot)**2)/effsteps
        totu = gusqrd - ((gutot)**2)/effsteps
        sigz = sqrt(totz/(effsteps - 1))
        sigu = sqrt(totu/(effsteps - 1))
        if (gmethod .eq. 2) then
          avgz = gzinput
          sigz = 0.0
        endif

        uex = -uint(avgz)
        cex = cap(avgz)
```

```
      write(gfullu,*) gxsize,t,avgz,t,sigz,t,uex,t,avgu,t,sigu,t,cex
    return
    end


c   Uses Newton's method to calculate the
c   inverse temperature from the avg demon
c   value.

    function findz(avgen,noofbins)
      real avgen,x0,xn,temp1,temp2,temp3,temp4,fx0,fprimex0
      integer noofbins,i,j
      x0 = 0.5
      epsilon = 0.00005
      do 20 i = 1,100

        temp1 = 0.0
        temp2 = 0.0
        temp3 = 0.0
        temp4 = 0.0

        do 10 j = 0,noofbins-1
          temp1 = exp(-4 * j * x0)
          temp2 = temp2 + temp1
          temp3 = temp3 + (j * temp1)
          temp4 = temp4 + (j * j * temp1)
 10     continue

        fx0 = (temp3/temp2) - avgen
        fprimex0 = (1 - (16 * temp4 * temp2))/(temp2 * temp2)

        xn = x0 - (fx0/fprimex0)
        if (abs(xn - x0) .lt. epsilon) goto 100
        x0 = xn

 20   continue

100   findz = xn
    return
    end


c   Calculates the exact value of the
c   internal energy, U, for a given Z.

    function uint(v)
      real v
      integer n,l
      if (v .le. 5.0) then
        n = 5
        temp1 = u(2.0*v,n)
        do 10 l = 1,12
          n = 2*n
          temp2 = u(2.0*v,n)
          if (abs(temp1-temp2) .lt. 0.00005) goto 100
          if (l .ne. 12) temp1 = temp2
 10     continue
100     uint = -temp2
```

```
        else
          uint = -2.0
        endif
      return
      end


c     Function called by uint(v).

      function u(w,n)
        real k,k1,pi,sum,h,t1
        integer ctn,i
        if (w .gt. 0) then
          pi = 3.1415926
          sum = 0.0
          ctn=2*n
          h = pi/ctn
          k1 = 2*sinh(w)/(cosh(w)**2)
          do 10 i = 1,n-1
            sum = sum + 1.0/sqrt(1.0 - (k1**2*(sin(i*h)**2)))
   10     continue
          sum = sum + 0.5*(1 + 1./sqrt(1-k1**2))
          k= h*sum
          t1 = 1 + (2*tanh(w)**2 - 1)*(2./pi)*k
          u = t1/tanh(w)
        else
          u = 0
        endif
      return
      end


c     Calculates the exact value of the
c     specific heat, C, for a given Z.

      function cap(v)
        integer n,l
        real v,temp1,temp2
        n = 20
        temp1 = C(2.*v,n)
        do 20 l = 1,12
          n = 2*n
          temp2 = C(2.*v,n)
          if (abs(temp1-temp2) .lt. .00005) then
            goto 10
          end if
          if (l .ne. 12) temp1 = temp2
   20   continue
   10   cap=temp2
      return
      end


c     Function called by cap(v).

      function C(w,n)
      real w,K,E,k1,pi,sum1,sum2,h,t1,t2
      integer n,i
```

```
      if (w .eq. 0) then
        C = 0
        return
      end if
      pi = 3.1415926
      k1 = 2*sinh(w)/(cosh(w)**2)
      sum1 = 0.0
      sum2 = 0.0
      h = pi/(2*n)
      do 10, i = 1,n-1
        sum1 = sum1 + 1.0/sqrt(1. - (k1**2*(sin(i*h)**2)))
        sum2 = sum2 + sqrt(1. - (k1**2*(sin(i*h)**2)))
10    continue
      sum1 = sum1 + .5*(1 + 1./sqrt(1-k1**2))
      sum2 = sum2 + .5*(1 + sqrt(1-k1**2))
      K = h*sum1
      E = h*sum2
      t1 = 1 + (2*tanh(w)**2 - 1)*(2./pi)*K
      t2 = 2*(K-E-0.5*pi*(1-tanh(w)**2)*t1)
      C = (.5/pi)*(w/tanh(w))**2*t2
    return
    end
```

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
BERKELEY, CALIFORNIA 94720